Assignment 1, Part - 1

Lua GC Report

By Sourodeep Datta, 21CS10064

1 Garbage Collection in Lua

In Lua, memory is managed automatically by running a garbage collector to collect all *dead* objects. An object is considered *dead* as soon as the collector can be sure the object will not be accessed again in the normal execution of the program. ("Normal execution" here excludes finalizers and operations using the debug library.)

Lua will not collect an object that may still be accessed in the normal execution of the program, and it will eventually collect an object that is inaccessible from Lua. (Here, inaccessible from Lua means that neither a variable nor another live object refer to the object.)

Lua never collects objects accessible through the registry, which includes the global environment. The registry is a predefined table that can be used by any C code to store whatever Lua values it needs to store.

1.1. Finalizers and Resurrection

Finalizers are metamethods which are called when the garbage collector detects that the corresponding table or userdata is dead. Finalizers help to coordinate Lua's garbage collection with external resource management such as closing files, network or database connections, or freeing memory.

For an object to be finalized when collected, it must be marked for finalization. This can be done by setting its metatable to a metatable which has its __gc metamethod set.

When a marked object becomes dead, it is not collected immediately by the garbage collector. Instead, Lua puts it in a list. After the collection, Lua goes through that list. For each object in the list, it checks the object's __gc metamethod: If it is present, Lua calls it with the object as its single argument.

Because the object being collected must still be used by the finalizer, that object (and other objects accessible only through it) must be *resurrected* by Lua. Usually, this resurrection is transient, and the object memory is freed in the next garbage-collection cycle. However, if the finalizer stores the object in some global place (e.g., a global variable), then the resurrection is permanent.

2 Incremental Garbage Collection

This is the default mode of the Lua GC (Garbage Collector).
In incremental mode, each GC cycle performs a mark-and-sweep collection in steps that are

interleaved with the program's execution. In this mode, the collector uses three numbers to control its garbage-collection cycles: the garbage-collector pause, the garbage-collector step multiplier, and the garbage-collector step size.

In Lua, this GC is run incrementally i.e. the program and GC run alternately. Whether or not GC happens is dependent on how much memory has been allocated since its last iteration.

2.1. Mark and Sweep

In mark and sweep, there are 2 steps:

- A marking step where the collector traverses through the program and marks objects to collect and leave based on what kind of references exist to those objects.
- A sweeping step where objects which were marked for collection are collected, and the GC process is completed.

2.2. GC Variables

There are 3 modifiable variables for this GC:

- garbage-collector pause: Controls how long the collector waits before starting a new cycle. The collector starts a new cycle when the use of memory hits n% of the use after the previous collection. For values less than or equal to 100, the collector starts the next cycle immediately.
- garbage-collector step multiplier: Controls the speed of the collector relative to memory allocation, that is, how many elements it marks or sweeps for each kilobyte of memory allocated. Larger values make the collector more aggressive but also increase the size of each incremental step.
- garbage-collector step size: Controls the size of each incremental step, specifically how many bytes the interpreter allocates before performing a step. This parameter is logarithmic: A value of n means the interpreter will allocate 2n bytes between steps and perform equivalent work during the step. A large value (e.g., 60) makes the collector a stop-the-world (non-incremental) collector.

2.3. Code Analysis

When running *incrementalgc*, there will be some calls made to *luaC_checkGC*. This function is called by functions which do operations that can lead to allocation of memory, such as *lua_pushlstring*, *lua_createtable* and *lua_newthread*.
luaC_checkGC is a macro for *luaC_condGC*.
luaC_condGC is a macro for *luaC_step*, which is the actual function that gets called. It performs a basic GC step if collector is running.

Based on the GC mode, either an *incstep* or a *genstep* is done. In this case, an *incstep* is done. In this function, the step multiplier is retrieved and used to calculated how much "debt" the GC has and how much "work" it has to do. The "debt" is proportional to the amount of memory the Lua state has accumulated.
It then enters a loop where is performs single steps of garbage collection by calling *singlestep*.

The amount of work done in each step is subtracted from the debt until the debt has been sufficiently reduced. The *singlestep* function may be named as *singlestep.lto_priv.0* in the call graph due to Link Time Optimization.

The *singlestep* function initially checks the state of the GC and determines whether a new collection cycle is being started or the previous one is being continued. It also initializes the amount of work done. During a collection cycle, the GC cycles through multiple states:

1. GCSpropagate: The GC marks objects by calling *propagatemark* until there are no more potential objects left to mark.

2. GCSenteratomic: The GC finalizes the markings.

3. GCSswpallgc / GCSswpfinobj / GCSswptobefnz ..: The GC sweeps regular objects, objects with finalizers, objects to be finalized etc. in these states. This is done by calling *sweepstep*.

3 Generational Garbage Collection

In generational mode, the GC does frequent minor collections, which traverses only objects recently created. If after a minor collection the use of memory is still above a limit, the collector does a stop-the-world major collection, which traverses all objects. The generational mode uses two parameters: the minor multiplier and the the major multiplier.

3.1. GC Variables

There are 2 modifiable variables for this GC:

- minor multiplier: Controls the frequency of minor collections. For a minor multiplier x, a new minor collection will be done when memory grows x% larger than the memory in use after the previous major collection.

- major multiplier: Controls the frequency of major collections. For a major multiplier x, a new major collection will be done when memory grows x% larger than the memory in use after the previous major collection.

3.2. Code Analysis

The first call in main for the GC goes to *lua_gc*. This is to change the mode of the GC from the default (incremental) to generational. Within this function, the minor and major multipliers are retrieved.
A check is done via *isdecGCmodegen* for the the mode of the GC. During this check, if in generational mode, it is possible that the collector can temporarily go into incremental mode to improve performance.
After this *luaC_changemod* is called. Here the GC mode is changed by calling *entergen*.
Within *entergen*, it first does a full collection cycle of the GC in incremental mode, by marking all objects using *atomic* and then transitioning to generation GC using *atomic2gen*. *atomic2gen* sweeps through the objects and removes all dead objects, and adds any surviving objects to the old generation.

After setting up the GC, the GC runs similar to incremental mode. Functions which do

operations that can lead to allocation of memory, such as *lua_pushlstring*, *lua_createtable* and *lua_newthread* make calls to *luaC_checkGC*.
luaC_checkGC is a macro for *luaC_condGC*.
luaC_condGC is a macro for *luaC_step*, which is the actual function that gets called. It performs a basic GC step if collector is running.

As the GC mode is generational, a call to *genstep* is done. This function does a "generational step". Usually this means doing a minor collection, which is done by *youngcollection*. If memory grows "major multiplier"% larger than it was at the end of the last major collection, the function does a major collection. This is done by fullgen. At the end, it checks whether the major collection was able to free a decent amount of memory (at least half the growth in memory since previous major collection). If so, the collector keeps its state, and the next collection will probably be minor again. Otherwise, it is a "bad collection". In this case, next collection will go to *stepgenfull*.

When the program is building a big structure, it allocates lots of memory but generates very little garbage. In those scenarios, the generational mode wastes time doing small collections, and major collections are frequently a "bad collection", a collection that frees too few objects. To avoid the cost of switching between generational mode and the incremental mode needed for full (major) collections, the collector tries to stay in incremental mode after a bad collection, and to switch back to generational mode only after a "good" collection (one that traverses less than 9/8 objects of the previous one).

4 Full Garbage Collection

Full Garbage Collection (or FullGC) refers to the function *luaC_fullgc* and the results of its invocation. *luaC_fullgc* can only be called by *lua_gc* with the *LUA_GCCOLLECT* flag. This function performs a full GC cycle, in the mode that the GC is currently set to (either incremental or generational). In this assignment, calls to this function have been made while the GC is set to its default mode (incremental). In incremental mode, a call is made to *fullinc* which then does the cycle as according to the what is mentioned here.